## 导言

机器做更多，人做更少，是工业时代不停追逐的发展趋势。而在软件测试中我们也希望将测试人员从重复性高，复杂度大的测试工作中解放出来，所以有了对于android测试的自动化尝试。本篇论文所关注的，即是具体的实现和分析——从android事件流到实际执行并产出测试产品。

## Android交互实现分析

从IFML演化出的交互事件流进而形成的自动化测试，在另一个角度上，是对可预期的用户行为的替代实现，换言之，自动化测试很大程度上是在模拟用户操作。而在android领域中，用户行为，或者说人机交互的种类，是简单而清晰的。无论应用的复杂度有多高，最终都是由几类基本用户行为实现交互。而这几类基本用户行为大致如下：

Click，点击: 包含基本的单次点击，多次点击（多为双次点击），以及长点击

Swipe，拉拖：包含单点单向拉拖，多点单向拉拖，单点变向拉拖，多点多向拉拖

Text，键入值：在不考虑手机键盘操作的情况下，仅仅是键入的值；当考虑键盘操作时，依旧可以拆解为click与swipe的组合

Device，设备：指手机所有功能键，包含基本的back、home、menu、volume等以及相机、电量、网络等及其组合

这四类基本操作的实现需要对象的存在，android中的组件对象也是分类清晰的，大致如下：

<**android组件对象分析**>

至此我们可以将一个功能的实现拆解为一连串的基本用户行为，而基本用户行为，可以用operation+component的形式表示，进一步的我们有了事件流的表示规则。

XML下的事件流格式

<project>

<properties>

</properties>

<eventsflow>

<!--scope>

</scope-->

<event>

<name></name>

<component>

<type></type>

<name></name>

</component>

<operation>

<type></type>

<!-- operation cas -->

<reference></reference>

<!-- optional -->

<datalist>

<data>

</data>

</datalist>

</operation>

</event>

<eventsflow>

</project>

## 测试的机器实现准备

在一致的事件流规则下，机器实现变得简单。但考虑到从底层实现click、swipe等操作模拟的可能复杂性以及目前已存在的android测试框架使用上的成熟性，最后选择使用目前的主流框架，来向实现靠拢。也因此有了以下的内容：

测试框架比较分析

### Android测试框架

目前比较主流的框架基本都是在Junit的基础上实现的，因为发行方的不同，不同框架在实现机制和性能侧重上各有不同。其中最基础的测试工具应当属Android APIs本身提供的Junit Testcase/Instrumentation Testcase类和Android提供的UIAutomator框架，以及Android SDK提供的Monkey Runner。以上几个工具是其它框架的基础，以下将沿着这些基础框架来进行比较。



#### Instrumentation系

基于Instrumentation类

通过把测试代码和应用代码，确切地说是测试APK和被测APK，运行在同一个进程中，通过Java反射机制，来获取当前窗口所有视图，并根据该视图查找到目标控件的属性信息，并计算出目标控件中心点坐标。然后，利用Instrument内部接口，实现点击操作。

Instrumentation类及拓展框架

Instrumentation类实际上是Android APIs的一部分，我们甚至可以将其理解为不依赖UI的Activity。Instrumentation通过向Android程序发送事件来驱动被测试程序达到测试Activity的目的，而不同于受UI事件驱动的UIAutomator系的测试框架。这种通过在测试用例中对Instrumentation和Activity直接指定操作的完全白盒的实现方式也有一定的弊端，是用apk去测apk，在对用户事件模拟上就有很大的不足，另外测试/待测程序在同一进程中运行，它的进程安全也受到限制，跨进程也无法实现。

Robotium框架

在Instrumentation的基础上增加了手势、webview、截图等等的支持，通过对用户在UI层操作动作的模拟，完成了黑盒+白盒的测试实现，也有更丰富的开发接口。但solo对象承担绝大部分的操作方法，导致实际使用中压力很大，面向对象方面较差。同时Robotium并未解决Instrumentation在进程上的不足，也延续了用app测app的方式，不支持跨进程测试。

CalaBash框架

Calabash是一套Android/IOS通用的测试框架，测试描述于Cucumber，然后在运行时转化为Robotium（Android）或Frank（IOS）。在设计上，Calabash更倾向用户使用，支持多种语言的指令和控制器。

Espresso框架

未找到源码，也是Google提供的实现，更加关注于UI本身，和UI线程同步，性能和稳定性表现优异，但除了UI之外几乎不涉及其它功能。

UIAutomator系

基于UI事件的测试实现。通过Android提供的各种服务，来获取当前窗口的视图信息。然后，在当前视图内查找目标控件，并根据该控件属性信息计算出该控件中心点的坐标，进而构造出一个Android Input事件来实现对应用的自动化测试。其主要特点是：测试代码和被测应用各自运行在各自的进程内，相互独立。

UIAutomator框架

基于UI的实现，实现方式的不同是UIAutomator相比于Instrumentation系所表现出的不同特性的根本原因。UIAutomator是一种黑盒测试，完全基于UI的实现方式也使它只需要关注UI层的活动而不需要关注具体进程的信息，所以跨进程测试（多app间切换）是它的优势。除优势之外，这种不涉及底层的测试方式也有很多弊端，首先是debug的难度较大，也无法获取当前load的Activity，更无法测试网络连接/声音/电量等等底层的程序功能和设备状态。

Appium框架

Appium是基于UIAutomator实现的，更像是UIAutomator丰富接口的封装，可以使用不同语言来编写测试代码，也可以测试不同平台程序：Android/iOS/Web/Hybrid

MonkeyRunner系（Monkey）

MonkeyRunner在实现上，不同于前俩类的实现依赖，而更靠近Android设备的直接控制。

未找到MonkeyRunner的源码，从网上的分析看，其和目标设备的通信实现方式是通过ChimpChat层进行封装分发但最终是在ddmlib进行处理的，其中囊括的方法大体如下:

1.发送monkey命令 ：MonkeyRunner先通过adb shell发送命令"monkey -port 12345"在目标机器上启动monkey以监听端口接受连接，然后MonkeyRunner通过连接该端口建立socket并发送monkey命令。 所有与界面相关的操作都是通过这种方式发送到目标机器的。

2.发送adb协议请求 ：通过发送adb协议请求来与目标设备通信的，其实adb命令行客户端的所有命令最终都是通过发送遵循adb协议的请求来实现的，只是做成命令行方式方便终端用户使用而已

3.发送adb shell命令 ：模拟adb命令行工具发送adb shell命令，只是 不是真正的直接命令行调用adb工具，而是在每一个命令执行之前先通过上面的“ 发送adb协议请求 “发送“shell：”请求建立一个和adb服务器通信的adb shell的socket连接通道，adb服务器再和目标设备的adb守护进程进行通信

这种直接和设备通信的实现方式，也使MonkeyRunner变成黑盒测试，在获取设备信息上，MonkeyRunner肯定有得天独厚的优势，但在测试的直观性上不如UIAutomator，在对程序的理解上不如Instrumentation。

## 测试的机器实现

在工具的综合考量下，选择了以Robotium为主，UIAotumator和appium为辅的测试实现



<**实现的问题**>

<**详细设计**>



## 总结

功能性测试————很多框架的功能不需要使用，只需要关注基本的用户操作

测试用例的实现